**Introduction**

This project is an implementation of a Bank Account Management System. The program consists of module “Account” and “Account Manager”, which contains their respective class. It also has a user interface handling user interactions, which has its own separate module.

Class Account is responsible for storing details of an account, which are account number, title, and balance. Account number is stored in form of integer, while account title is stored in a string. For balance, it is stored in float for precision. The class Account also handles withdrawal and deposit.

Class Account Manager stores multiple instances of class Account, handles operations such as account creation, deletion, searching, balance checking, and traversal. Accounts are stored in Binary Search Tree data structure for quick account lookup.

The user interface is a command line interface that allows user to create and manage multiple accounts within an account manager. Users can interact with the account manager with multiple commands, such as add\_account, check\_balance, and so on. It also supports command aliases to enhance the ease of user interactions. Most exceptions are also handled gracefully in this module without crashing the program.

**Dependencies**

The AccountManager module depends on Account and the module for Binary Search Tree to operate, as it is relied to define and store accounts. Binary Search Tree in turn relies on Queue module in order to save traversal order, while Queue module relies on Linked List module to store its element.

As for user interface, other than depending on Account and AccountManager, it also utilizes readline library to provide support for command history for convenience of users.

**Terminologies**

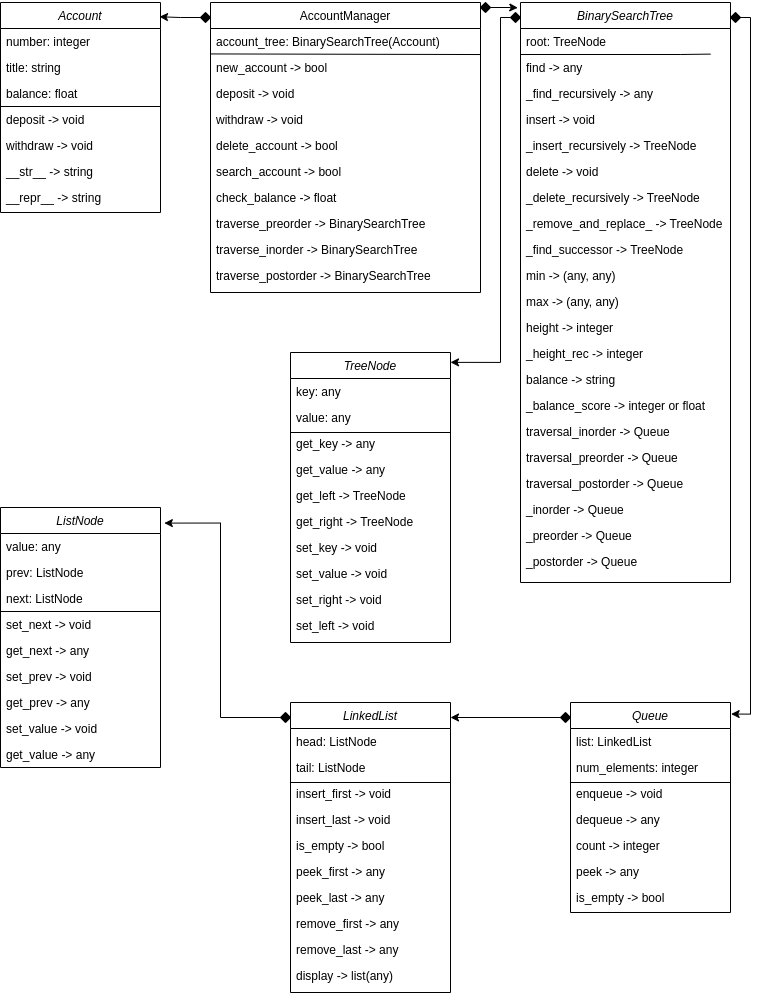
|  |  |
| --- | --- |
| Term | Meaning |
| Acc | Account |
| Num | Number |
| Args | Arguments |
| Cmd | Command |

**Future Directions**

Account Manager could be improved by implementing more banking related functionalities, such as interest rate and balance transfer. Accounts data could also be saved to and loaded from a file or cloud storage at the end and the start of the program. Account operations such as deposit and withdrawal might also require password or credential that is stored in an encrypted file, allowing for more secure operation.

For the user interface, it could be improved by allowing auto-completion of commands and miscellaneous functionalities such as access control to store and display specific list of accounts for respective users for seamless experience.

**Class UML Diagrams**

****

**Complexity Analysis**

All analysis is based on class AccountManager, because it serves as a front end for other classes. As for user interface, most of its operations utilizes AccountManager directly.

Method new\_account:

account creation → O(1)

* Reason: account creation is performed in one step.

account insertion → Best case: O(log n), Worst case: O(n)

* Reason: BST operation divides and conquers when the tree is balanced resulting in time complexity of O(log n). However, the tree can degenerate to linked list, making access time O(n) at the worst case.

find account → Best case: O(log n), Worst case: O(n)

* Reason: BST search operation

Method deposit:

find account → Best case: O(log n), Worst case: O(n)

* Reason: BST search operation

deposit → O(1)

* Reason: Depositing is performed in one step.

Method withdraw:

find account → Best case: O(log n), Worst case: O(n)

* Reason: BST search operation

withdraw → O(1)

* Reason: Withdrawal is performed in one step.

Method delete\_account:

find account → Best case: O(log n), Worst case: O(n)

* Reason: BST search operation

delete account → Best case: O(log n), Worst case: O(n)

* Reason: BST node deletion operation

Method search\_account:

find account → Best case: O(log n), Worst case: O(n)

* Reason: BST search operation

print details → O(1)

* Printing is performed in one step.

Method check\_balance:

find account → Best case: O(log n), Worst case: O(n)

* Reason: BST search operation

print balance → (1)

* Printing is performed in one step.

All methods of traversal:

O(n) Guaranteed

* Reason: Each node is traversed in O(1) and is enqueued to the traversal queue in O(1), which grows the linked list used to store queue elements in O(1). All of this makes the operation O(3n) but constant values are ignored, becoming O(n)

Module Interface

Traversal printing:

O(n) Guaranteed

* Reason: The traversal queue is dequeued until empty, each dequeue is O(n)

Help message construction:

O(n)

* Reason: Each command is appended to help message in O(1)

Command argument validation:

O(1)

* Reason: validation is done in one step by passing command to key dict and receiving its argument count to compare with the actual number of arguments.

User input command matching:

Worst case: O(n) \* average number of command aliases

Best case: O(2)

* Reason: user input is checked against all commands, and each command, user input is also checked against its aliases. The number of aliases for each command is not fixed.

**Traceability Matrix**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | Requirement | Test Case | Implementation | Status |
| 1. | Define account class | test\_constructor()  test\_deposit()  test\_withdraw() | \_\_init\_\_()  deposit()  withdraw() | Pass  Pass  Pass |
| 2. | Manage account using BST | test\_new\_account()  test\_deposit()  test\_withdraw()  test\_delete\_account() | new\_account()  deposit()  withdraw()  delete\_account() | Pass  Pass  Pass  Pass |
| 3. | Searching for account | test\_search\_account() | search\_account() | Pass |
| 4. | Check balance | test\_check\_balance() | check\_balance() | Pass |
| 5. | Traversal of account records | test\_traverse\_preorder()  test\_traverse\_inorder()  test\_traverse\_postorder() | traverse\_preorder()  traverse\_inorder()  traverse\_postorder() | Pass  Pass  Pass |
| 6. | Interactive Interface and User Interaction | Manual testing | interface.py | Pass |